36p.

Semi-Annual Progress Report

Contract No. NAG-1-260 IN-31463

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmund H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

S. T. Gregory
Graduate Research Assistant

J. I. A. Urquhart
Graduate Research Assistant

Report No. UVA/528213/CS86/107

August 1985

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

(NASA-CR-179806) THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS Semiannual Progress Report (Virginia Univ.) 36 p CSCL 09B G3/61 44681

N87-12243

Unclas

Semi-Annual Progress Report

Contract No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmund H. Senn
ACD, MS 125

Submitted by:

J. C. Knight
Associate Professor

S. T. Gregory
Graduate Research Assistant

J. I. A. Urquhart
Graduate Research Assistant

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/CS86/107 Copy No. _____
August 1985

1. Introduction

The purpose of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

During this grant reporting period our primary activities have been:

* Ada is a trademark of the U.S. Department of Defense

(1) Continued development and testing of our fault-tolerant Ada testbed.

(2) Development of suggested changes to Ada so that it might more easily cope with the failures of interest.

(3) Design of new approaches to fault-tolerant software in real-time systems, and the integration of these ideas into Ada.

(4) The preparation of various papers and presentations.

The various implementation activities of our fault-tolerant Ada testbed are described in section 2. In our analysis of the deficiencies of Ada, it has been quite natural to consider what changes could be made to Ada to allow it to have adequate semantics for handling failure. In section 3, we describe some thoughts on this matter reflecting what we consider to be the minimal changes that should be incorporated into Ada.

We consider it to be important that attention be paid to software fault tolerance as well as hardware fault tolerance. The reliability of a system depends on the correct operation of the software as well as the hardware. Our concerns in this area are discussed in section 4.

During this grant reporting period we have made various reports about this work. Our activities in this area are described in section 5.

A list of papers and reports prepared under this grant, other than the annual and semiannual progress reports, is presented in Appendix 1. A

paper presented during this grant reporting period is included as Appendix 2 (see also section 4).

2. Implementation Status

At the beginning of this grant reporting period a major activity that we undertook was to move the entire testbed to a network of Apollo DN300 workstations connected via a local area network. This was a substantial undertaking since the Apollo and the VAX systems are quite different. We set as a goal the development of a single version of the source text of the testbed which would reside on the VAX. The Apollo version would be build from this by various filters and other modification programs. The benefit of this approach is that a single system exists (in principle) and so bugs can be fixed and other changes made in one place but take effect on both computer systems. We achieved this goal and have ported successfully the entire testbed (but not the translator) to the Apollo system. The simple test cases that we have executed on the VAX implementation operate correctly on the Apollos.

One aspect of the translation of the system to the Apollos that surprised us was the Apollo system's communication performance. The VAX version of the testbed implements a virtual network using UNIX "pipes". Clearly since the Apollo implementation uses real processors and a real communication system, it was necessary to replace this part of the testbed. This was anticipated and performed. The Apollo communication system is a proprietary token-ring bus operating at 10 Mhz. Unfortunately this network is not directly available to an application program. We have been promised direct access on a restricted basis by Apollo Corporation on many occasions but this access has not been forthcoming.

The interface provided by the Apollo operating system is message based but requires one of the network nodes to be executing a message-switching program, thus implementing a "star" network. Further, all communication between nodes in this logical star is written to a disk and read from that disk on its way from one node to another. This slows the effective communication rate down by several orders of magnitude. Apart from the difficulty that we experienced in determining exactly how this interface worked and how to use it, when we finally got the necessary communication operating, we discovered transmission speeds that are of the same order as RS232.

During this grant reporting period there have been numerous changes to the computer systems that we use that have affected our Ada testbed. They are:

(1) The version of UNIX used by our VAX was changed from 4.1BSD to 4.2BSD. The testbed makes extensive use of the tasking features of UNIX and other system facilities for terminal access and control. Much of this interface was changed in the transition to 4.2BSD and the testbed was not operational initially under 4.2BSD. All the necessary changes have now been made and the testbed now operates correctly under 4.2BSD.

(2) The disk system for the Apollo network was enhanced substantially. Several new, small disk units were added to improve performance of the various Apollo workstations. In making this enhancement, subtle

changes to the file system (normally transparent) were made and this affected the testbed in a number of ways. All the modifications needed to cope with the changes have now been made in the testbed.

(3) The operating system used by the Apollo network was upgraded to take advantage of new facilities offered by Apollo. This has had some subtle effects on the services that the testbed uses and stopped the testbed from operating on the Apollos for some time. We have made the necessary changes to the testbed to cope with the operating system changes.

Given the poor communication performance offered by the Apollo network we question the utility of having the testbed available on that network. Our department expects to receive equipment to implement a new network from a different supplier and we are considering using the new equipment rather than continuing to use the Apollos.

We have very little control over the poor communications facilities provided to us despite the fact that they play a significant role in the testbed's overall performance. However, in testing the testbed, we have discovered that it is extremely inefficient in its own right. We never intended the testbed to be an efficient implementation since our primary concern was functionality. However, in trying to run test programs, we have been frustrated by the slowness of the implementation. Consequently, we have started a program of improving its efficiency in which we are examining the algorithms and data structures used in the testbed. At

present we have nothing to report by way of performance improvement. However, our initial investigation suggests that there are numerous areas that can be enhanced. We expect to report on the efficiency gains in our final report for 1985.

During this grant reporting period we have extended the translator for the subset of Ada that interests us and the translator is now essentially complete. Although it operates on the VAX, it must be kept in mind that it generates code for a synthetic Ada machine and so its output can be used by the testbed when operating on any target, in particular the Apollo network. The code quality produced by the translator is quite poor since, once again, efficiency is not a concern of this project. However, in concert with our efforts to improve the efficiency of the testbed, we are considering improving the code quality produced by the translator.

3. Ada And Hardware Fault Tolerance

We have summarized our concerns about Ada's inability to deal with processor failure by pointing out that the problem is basically one of omitted semantics. Nothing is stated in the Ada Language Reference Manual about how programs are to proceed when a processor is lost in a distributed system although the manual does include distributed computers as valid targets specifically.

We have proposed additional semantics to deal with this situation. The heart of these additional semantics is the notion that the loss of a processor and consequently the loss of part of the program can be viewed as equivalent to the execution of abort statements on the lost tasks. Thus in all cases, failure semantics would be equivalent to the semantics of abort. We have also proposed a comprehensive mechanism for implementing these semantics. This mechanism requires quite extensive changes to the execution-time support for Ada but it is feasible as we have shown in our testbed implementation.

The use of abort semantics is not an elegant approach. There are numerous consequences that seem rather extreme if considered out of context. For example, abort semantics imply that all the dependent tasks of a task that is lost must be terminated even if they are still executing on non-failed computers. The *overwhelming* advantage of abort semantics is that they do not require that the language be changed.

A more elegant and clearly preferable approach in the long run is to modify the language and to introduce language structures that include appropriate failure semantics. During this grant reporting period we have continued our consideration of what form these language structures might take. Our conclusions are contained in a language design that we call Ada-2. Some of the general aspects of Ada-2 will be summarized here briefly, and a complete description supplied to the sponsor under separate cover.

In Ada-2, the sequential part of Ada is left unchanged except for rules of visibility. However, those parts of Ada that deal with tasks and the concept of a program have been changed completely. Packages have been slightly modified and play a more important role than before.

The main structural element in many sequential languages is the procedure. If procedures could be carried over to the distributed case, then the distribution could be made transparent to the programmer. This is an attractive idea and various attempts have been made to achieve it. Nelson's work [1] on remote procedure calls is a typical example. Even in Ada, communication with a task was designed to appear to the caller like a procedure call. However, in our opinion, the procedure may not be a good model for distributed computing.

To understand why, it is useful to consider which of those properties of a procedure in a sequential language carry over naturally to the distributed case. In the sequential case a procedure is subordinate to its caller. At the point of call, provision is made for the return of values, an instance of a

procedure is created, and values are passed to it. At this point the caller, called the creator process, turns over control of its run-time environment to the procedure which executes and then returns control to the creating process.

A first step towards the distributed case is to let a separate process, called the procedure process, be created to run the procedure when a remote procedure call is made. This is not efficient but conceptually nothing is really changed. As before, the creator of the procedure process remains suspended until the procedure process has completed its execution.

A further step towards the distributed case is to imagine that the creator process and the procedure process are on different machines. If the creator process remains suspended while the procedure process runs, then logically this remains close to the original sequential picture. However, if the processes are on different machines the only reason for suspending the creator process is to *mimic* the sequential case. From the viewpoint of the distributed case the natural thing is to let both processes proceed. This is substantially different from the sequential case.

If both tasks proceed, something must be done about the return of values to the creating process. Again the natural thing is for the creating process to make provision for the return at the time of the call; the creating process must then be able to obtain results or wait for results at any subsequent time in its execution. It is important to notice that the procedure process here is still subordinate to the creator process; it is created, performs some function, and is removed.

This model of a task has arisen through the natural extension of a procedure call to a distributed model. A quite different model of a task arises from the extension of the idea of a program to a distributed situation. Here each processor would be running a separate program. There is no creator and there are no subordinates, communication is not constrained to occur at the beginning and end of one of the processes, and communication with other processes can occur. This idea of a task is quite different from the idea of a procedure task and is closer to the idea of an Ada task. In fact Ada programs commonly use procedures, encapsulated in a package, to enforce some entry call protocol for a task also contained in the package. Such procedures are logically equivalent to a procedure task.

It might be argued that a procedure task is just a simple task and that there should not be separate units in the language to describe them. In fact the two concepts arise from different viewpoints and are used differently. In particular, the *failure semantics* for tasks and for procedure tasks are completely different; procedure tasks depend on their creator, tasks do not. In Ada-2 there are two flavors of tasks; tasks which model the procedure tasks described above and others that are similar to Ada tasks.

It is important to be able to collect various related run-time objects together and to provide an interface to them for the user. This is the reason that Ada has packages. However, in Ada packages, the different requirements that a programmer has for compile-time and for run-time objects are confused. This is very important for distributed programs since the distribution occurs at run time, not at compile time. Objects that exist

at compile time only need not be concerned with failure semantics but run-time objects must always be concerned with failure semantics. This has been missed completely in Ada.

The program unit that provides encapsulation in Ada-2 is also called a package. Ada-2 packages are similar to Ada packages, although the role of an Ada package which does not define run-time objects is taken by a *named declarative group*. The named declarative group encapsulates declarations which do not define run-time objects but which enable a unit using the declarative group to define run-time objects. Named declarative groups will thus contain type declarations for data, packages, tasks, sub-tasks and procedures. Named declarative groups can also contain declarations of constants. Unlike packages which can be considered to have a run-time existence, named declarative groups will not exist at run-time. and can be viewed as being *copied* across the various machines in a distributed system.

Both packages and named declarative groups can be used by units other than the declaring unit by being mentioned in a **with** statement. Declarations intended for use by the declaring unit only would be defined in un-named declarative groups; an un-named declarative group can contain declarations of run-time objects. As an un-named declarative group can only be used by the enclosing unit, un-named declarative groups will not be considered program units.

The remaining program units in Ada-2 are bodies of tasks, packages, sub-tasks and procedures. In contrast to Ada a body is not restricted to

appear in the same declarative part as the specification. A body, however, must occur before the declaration of an object which uses the body.

At the top level, an Ada-2 program consists of named declarative groups, packages and package bodies, and tasks and subtasks. The top level declarative groups are convenient for encapsulating global constants and types. All top-level packages must specify which processor they will run on. Execution of the program is initialized by starting the top-level packages on their designated processors in some arbitrary order. It is this program organization that permits a clear approach to failure semantics. The details of failure semantics are too lengthy to be described here.

4. Ada And Software Fault Tolerance

In a previous grant reporting period, we examined the literature on fault-tolerant software with the goal of determining the adequacy of Ada in providing a software fault tolerance mechanism. We found that Ada makes no provision for software fault tolerance. Consequently we have considered what extensions to Ada might be desirable to support fault-tolerant software.

Software fault tolerance is rarely used in practice and, when it is used, it is ad hoc with no formalism or organization. One of the reasons for this state of affairs is the general inadequacy of existing proposals for building software in a fault-tolerant manner. Before reviewing Ada and trying to incorporate software fault tolerance mechanisms into the language changes, we reviewed the state of the art and prepared a systematic set of criticisms of existing proposal for the provision of fault tolerance in software. Since our last grant report we have made considerable progress in defining new approaches to fault-tolerant software and integrating them into Ada. We have defined constructs that we call the *dialog* and the *dialog_sequence* which give new flexibility and control in the provision of backward error recovery. These constructs have been described in a paper that has been supplied to the sponsor under a separate cover.

We have pursued the implementation issues for these constructs during the current grant reporting period and completed an analysis of the implementation issues. Details will be supplied to the sponsor.

After analyzing the implementation issues to determine implementability, we turned our attention to the integration of these constructs into existing languages, particularly Ada. Unfortunately, we consider that we have isolated fundamental problems with existing programming languages, typified by Ada, that make integrating backward error recovery very difficult. The problems center around conflicts between the linguistic restrictions that have to be imposed by backward error recovery if it is to be successful, and the perceived needs of the programming language user. Our concerns are remarkably similar to those expressed by other workers who have reviewed programming languages with other objectives in mind. For example, the community of researchers interested in formal verification have found the same difficulties with languages like Ada as we do.

We have prepared a detailed report on the difficulties of integrating backward error recovery, and another report that addresses them and proposes various solutions. These reports will be supplied to the sponsor separately.

5. Professional Activities

During this grant reporting period we prepared and presented a paper about our work under this grant on fault-tolerant software at the Fifteenth International Symposium on Fault-Tolerant Computing held in Ann Arbor, Michigan. A copy of that paper as it appears in the Digest of Papers is included in this report as Appendix 2. We also gave a seminar describing the work at NASA's Goddard Space Flight Center.

REFERENCE

(1) B.J. Nelson, "Remote Procedure Call", PhD Dissertation, Computer Science Department, Carnegie Mellon University, May 1981.

Appendix 1

The following is a list of papers and reports, other than progess reports, prepared under this grant.

(1) Knight, J.C. and J.I.A. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings of the *AIAA Computers in Aerospace Conference*, October 1983, Hartford, CT.

(2) Knight, J.C. and J.I.A. Urquhart, "The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems", *Ada LETTERS*, Vol. 4 No. 3 November 1984.

(3) Knight, J.C. and J.I.A. Urquhart, "On The Implementation and Use of Ada on Fault-Tolerant Distributed Systems", submitted to *IEEE Transactions on Software Engineering*.

(4) Knight J.C. and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

(5) Knight J.C. and S.T. Gregory, "A New Linguistic Approach To Backward Error Recovery", Digest of Papers FTCS-15: *Fifteenth Annual Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI.

(6) Knight, J.C. and J.I.A. Urquhart, "Difficuties With Ada As A Language For Reliable Distributed Processing", Unpublished.

(7) Knight, J.C. and J.I.A. Urquhart, "Programming Language Requirements For Distributed Real-Time Systems Which Tolerate Processor Failure", Unpublished.

Appendix 2

This appendix contains the text of a paper prepared under this grant and presented at FTCS15: The Fifteenth International Symposium On Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985.

A NEW LINGUISTIC APPROACH TO BACKWARD ERROR RECOVERY*

Samuel T. Gregory John C. Knight**

Department of Computer Science
University of Virginia
Charlottesville, Virginia, U.S.A. 22903
(804) 924-7605

ABSTRACT

Issues involved in language facilities for backward error recovery in critical, real-time systems are examined. Previous proposals are found lacking. The dialog, a new building block for concurrent programs, and the colloquy, a new backward error recovery primitive, are introduced to remedy the situation. The previous proposals are shown to be special cases of the colloquy. Thus, the colloquy provides a general framework for describing backward error recovery in concurrent programs.

Subject Index:
Reliable Software - Interprocess Communication and Synchronization

*This work was sponsored by NASA grant number NAG1-260 and has been cleared for publication by the sponsoring organization.

**Presenter at FTCS-15 if paper is accepted.

1. INTRODUCTION

In this paper we examine the issues involved in the use of backward error recovery in critical, real-time systems. In particular, we are concerned with language facilities that allow programmers to specify how alternate algorithms are to be applied in the event that an error is detected. The best-known approach is the *conversation*[1]. Many difficulties with conversations have been pointed out including the lack of any time-out provision and the possibility of deserter processes. We introduce a new building block for concurrent programs called the *dialog* and a new backward-error-recovery primitive called the *colloquy* that remedy the various limitations of the conversation. The colloquy is constructed from dialogs and provides a general framework for describing backward error recovery in concurrent programs.

All of the syntactic proposals that we introduce are derived from Ada® [2]. The dialog and colloquy are proposed as general concepts but the specific syntax for their use is given as extensions to Ada. The actual syntax is irrelevant; the concepts could be used in many other programming languages. However, once chosen, a rigid syntax can allow a compiler to enforce certain of the semantic rules.

In section two, we briefly describe the concept of the conversation and the associated syntactic proposals that have been made. Issues that have been raised with conversations are discussed in section three. In section four, we present a syntax for the dialog called the *discuss* statement. In section five, we introduce the colloquy and a new statement called the *dialog_sequence* which allows the specification of the actions needed for a colloquy. In section six, we discuss the use of colloquys in the implementation of all previous approaches to backward error recovery.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2. CONVERSATIONS

The *conversation* is the canonical software fault–tolerance proposal for dealing with communicating processes. In a conversation a group of processes separately establish recovery points and begin communicating. At the end of their communication (i.e. the end of the conversation), which may include the passage of multiple distinct sets of information, they each wait for the others to arrive at an acceptance test for the group. If they pass the acceptance test, they *commit* to the information exchange that has transpired by discarding their recovery points and proceeding. Should they fail the acceptance test, they all restore their states from the recovery points. No process is allowed to *smuggle* information in or out by communicating with a process that is not participating in the conversation. Conversations can be nested; from the point of view of a surrounding conversation, a nested conversation is an atomic action[3].

Although not explicitly stated in the literature, it is assumed that if an error occurs during a conversation such that the acceptance test fails, the *same* set of conversant processes attempt to communicate again once individually rolled back and reconfigured (rather than proceeding on unrelated activities). It follows that they eventually reach the *same* acceptance test again. It is also presumed that any other failure of one of the processes is taken as equivalent to a failure of the acceptance test by all of them.

The processes in a conversation are the components of a system of processes. Error detection mechanisms for this system consist of announcement of failure by any one of the components and the single acceptance test. The acceptance test evaluates the combined states of the component processes with the designed intent of their communications. Damage assessment is complete *before* execution begins since the individual states of all the processes involved in the conversation are suspect, but no other processes are affected. Error recovery consists of restoring each process to the state it had as it entered the conversation, and the system of processes continues with its service by allowing each process to re–try

the communication perhaps using an alternate mechanism within that process for the communication activity.

Conversations were originally proposed as a structuring or design concept without any syntax that might allow enforcement of the rules. Russell[4] has proposed the "Name-Linked Recovery Block" as a syntax for conversations. The syntax appropriates that of the recovery block[5]. What would otherwise be a recovery block, becomes part of a conversation designated by a conversation identifier. The primary and alternate activities of the recovery block become that process' primary and alternate activities during the conversation, and the recovery block's acceptance test becomes that portion of the conversation's acceptance test appropriate to this process. The conversation's acceptance test is evaluated after the last conversant reaches the end of its primary or alternate. If any of the processes fail its acceptance test, all conversants are rolled back.

Kim has examined several more possible syntaxes for conversations[6]. His approaches assume the use of monitors[7] as the method of communication among processes. He examines the situation from two philosophies toward grouping. In one scheme, the conversing activities are grouped with their respective processes' source code, but are well marked at those locations. In another scheme, the conversing actions of the several processes are grouped into one place so that the conversation has a single location in the source code. The issue he is addressing is whether it is better to group the text of a conversation and scatter the text of a process or to group the text of a process and scatter the text of a conversation. A third scheme attempts to resolve the differences between the first two.

3. ISSUES WITH CONVERSATIONS

Desertion is the failure of a process to enter a conversation or arrive at the acceptance test when other processes expect its presence. Whether the process will never enter the

conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test, does not matter to the others if they have real–time deadlines to meet. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Whether they protect inter–process communications or sequential parts of processes, acceptance tests must be reached and reached on time for the results to be useful. Meeting real–time deadlines is as important to providing the specified service as is producing correct output. In order to deal effectively with desertion, especially in critical systems, some form of timing specification on communication and on sequential codes is vital.

When it needs to communicate, a process enters a conversation and stays there, perhaps through many alternate algorithms, until the communication is completed successfully. The same group of processes are required to be in the alternate interactions as were in the primary. The recovery action merely sets up the communication situation again. In the original form of conversation, once a process enters the construct, it cannot break out and *must* continue trying with the same set of other processes, including one or more which may be incapable of correct operation. In practice, when a process fails in a primary attempt at communication with *one group of processes* to achieve its goal, it may want to attempt to communicate with an *entirely different group* as an alternate strategy for achieving that goal; in fact, different processes might make different numbers of attempts at communicating. Conversations do not allow this, although it is not desertion if it is systematic and intended.

In a conversation, once individually rolled back and reconfigured, the same set of conversant processes attempt to communicate again, and eventually reach the *same* acceptance test again. True *independence* of algorithms between primary and alternates, within the context of backward error recovery, might require very *different* acceptance tests for each algorithm, particularly if some of them provide significantly degraded

services. A single test for achievement of a process' goal at a particular point in its text would of necessity have to be general enough to pass results of the most degraded algorithm. This might be too general to enable it to catch errors produced by other, more strict, algorithms. These considerations suggest the need for separate acceptance tests specifically tailored for each of the primary and alternate algorithms.

It must also be remembered, that although each process has its own reasons for participating, there is a goal for the *group* of processes as well. Rather than combine the individual goals of the many participants with the group goal in a single acceptance test (perhaps allowing the programmer to forget some), and rather than replicating the test for achievement of the group goal within every participant, there should be a separate acceptance test for each participant and another for the group.

A final problem with the conversation concept as it was originally defined, is that if a process runs out of alternates, no scheme is provided or mentioned for dealing with the situation.

4. THE DIALOG

We define a *dialog* to be an occurrence in which a set of processes:

(a) establish individual recovery points,

(b) communicate among themselves and with no others,

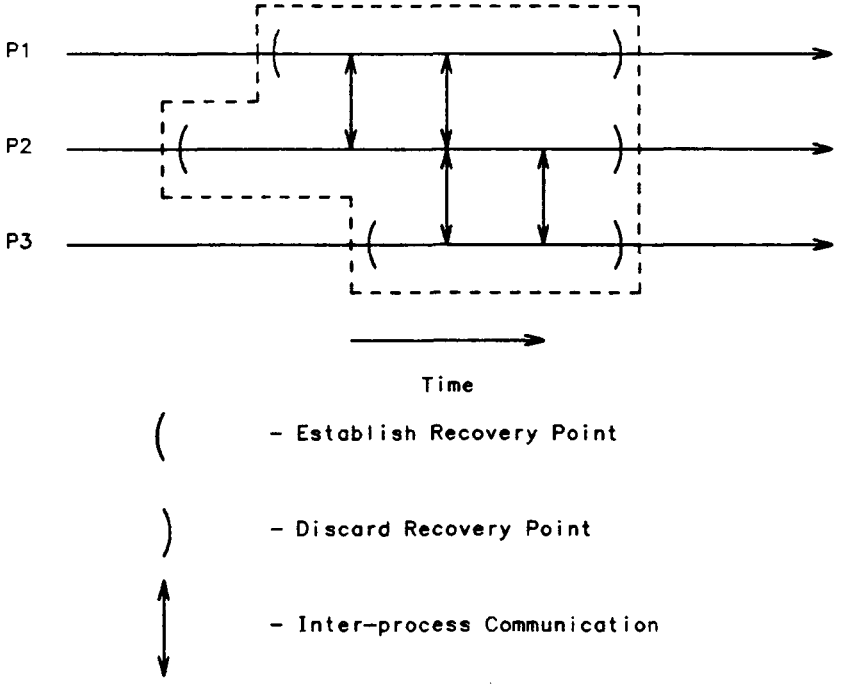
(c) determine whether all should discard their recovery points and proceed or restore their states from their recovery points and proceed, and

(d) follow this determination.

Success of a dialog is the determination that all participating processes should discard their recovery points and proceed. *Failure* of a dialog is the determination that they should restore their states from their recovery points and proceed. Nothing is said about what

should happen *after* success or failure; in either case the dialog is complete. Dialogs may be properly nested, in which case the set of processes participating in an inner dialog is a subset of those participating in the outer dialog. Success or failure of an inner dialog does not necessarily imply success or failure of the outer dialog. Figure 1 shows a set of three processes communicating within a dialog.

We introduce the *discuss* statement as a syntactic form that can be used to denote a dialog. Figure 2 shows the general form of a discuss statement. The *dialog_name* associates a particular discuss statement with the discuss statements of the other processes participating in this dialog, *dynamically* determining the constituents of the dialog. This association cannot in general be known statically. At execution time, when control enters a



Three Processes Communicating in a Dialog
Figure 1

```
DISCUSS dialog_name BY

    sequence_of_statements

TO ARRANGE Boolean_expression;
```

A DISCUSS Statement
Figure 2

process' discuss statement with a given dialog name, that process becomes a participant in a dialog. Other participants are any other processes which have already likewise entered discuss statements with the same dialog name and have not yet left, and any other processes which enter discuss statements with the same dialog name before this process leaves the dialog. Either all participants in a dialog leave it with their respective discuss statements successful, or all leave with them failed, i.e. the dialog succeeds or fails.

The *sequence of statements* in the discuss statement represent the actions which are this process' part of the group's actions within their dialog. Any inter-process communication *must take place within* this sequence of statements (i.e. be protected by a dialog). The discuss statement fails if an exception is raised within it, if an enclosed *dialog_sequence* (see below) fails, or if any timing constraint is violated.

The *Boolean_expression* is an acceptance test on the results of executing the sequence of statements. It represents the process' *local* goal for the interactions in the dialog. It is evaluated after execution of the sequence of statements. If this Boolean expression or that in the corresponding discuss statement of any other process participating in this dialog is evaluated **false**, the discuss statement of each participant in the dialog *fails*. If all of the local acceptance tests succeed, the common goal of the group, i.e. the *global* acceptance test is evaluated. If this common goal is **true**, the corresponding discuss statements of all participants in the dialog succeed; otherwise they fail. Syntactically, the common goal is

specified by a parameterless Boolean function with the same name as the dialog name in the discuss statement.

We stated that the participants in a particular dialog cannot be known statically. There may be, say, three processes whose texts contain references to a particular dialog name. If two of them enter a dialog using that name, questions might arise about participation of the third. The third process may be executing some other portion of its code so that it is unlikely to enter a dialog of that name in the near future. If the two processes reach and pass their acceptance tests, they, being the only participants in the dialog, can leave it -- the third process is not necessary to the dialog, so is not a deserter. If the dialog fails due to an acceptance test or a timeout (see below), the problem is not guaranteed to be the absence of the third process, so again it is not (necessarily) a deserter. If the dialog has no time limit specified (see below), that had to be by conscious effort of the programmer, so the two processes becoming "hung" in the dialog while waiting for the third was *not* unplanned.

The dialog names used in discuss statements are required to be declared in *dialog declarations*. The general form of a dialog declaration is:

```
DIALOG function_name SHARES ( name_list );
```

The *function_name* is the identifier being declared as a dialog name (and the name of the function defining the global acceptance test). The names mentioned in the *name_list* are the names of *shared* variables which will be used within dialogs that use this dialog name. This includes variables used within the function that implements the global acceptance test. Only a variable so named may be used within a discuss statement, and then only within discuss statements using a dialog name with that variable's name in its dialog declaration. The significance of these rules is that the set of shared variables can be locked by the compiler and execution-time support system to prevent smuggling. In effect, the actions of the dialog's participants are made to appear atomic to other processes with respect

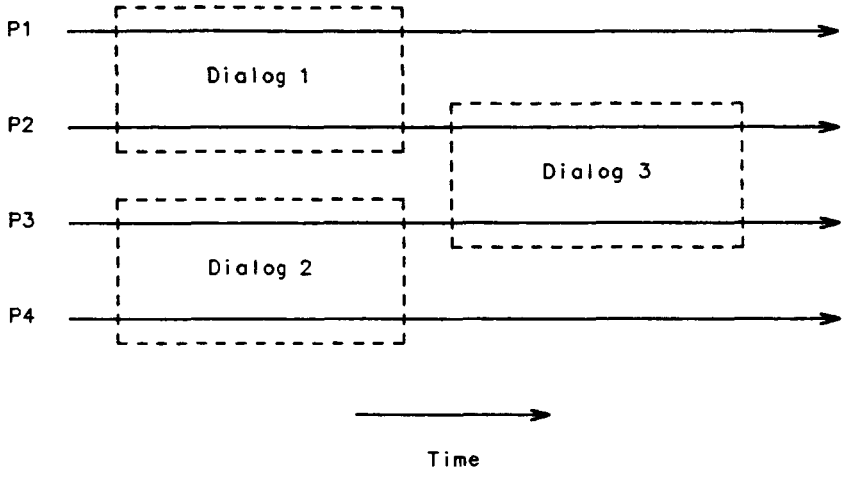
to these variables. (Our implementation, not described here, also prevents smuggling via messages or rendezvous).

The Boolean function named by the dialog name is evaluated after all processes in the dialog have evaluated their respective Boolean expressions and they all evaluate to **true**. It is only evaluated once for an instance of the dialog; i.e. it is not evaluated by each participating process. Thus no process can leave a dialog until all processes currently in that dialog leave with the same success, and success involves the execution of both a local and a global acceptance test.

5. THE COLLOQUY

A *colloquy* is a semantic construct that solves the problems of conversations. Unlike conversations, the rules of order and participation are well–defined and explicitly laid out.

A colloquy is a collection of dialogs. At execution time, a dialog is an interaction among processes. Each individual process has its own *local* goal for participating in a dialog, but the group has a larger *global* goal; usually providing some part of the service required of the entire system. If, for whatever reason, any of the local goals or the global goal is not achieved, a backward error recovery strategy calls for the actions of the particular dialog to be undone. In attempting to ensure continued service from the system, each process may make another attempt at achieving its original local goal, or some *modified* local goal through entry into a *different* dialog. Each of the former participants of the now defunct dialog may choose to interact with an *entirely separate* group of processes for its alternate algorithm. The altered constituency of the new dialog(s) most certainly requires new statement(s) of the original global goal. The set of dialogs which take place during these efforts on the processes' part is a *colloquy*. A set of four processes engaged in a colloquy that involves three dialogs is shown in Figure 3.



Four Processes in a Colloquy of Three Dialogs
Figure 3

A colloquy, like a dialog or a rendezvous in Ada, does not exist syntactically but is entirely an execution–time concept. The places where the text of a process statically announces entry into colloquys are marked by a variant of the Ada select statement called a *dialog_sequence.*

The general form of a dialog_sequence is shown in Figure 4. At execution time, when control reaches the **select** keyword, a recovery point is established for that process. The process then *attempts* to perform the activities represented in Figure 4 by `attempt_1`. The attempt is actually a discuss statement followed by a sequence of statements. To ensure proper nesting of dialogs and colloquys, a discuss statement may appear only in this context. If the performance of these activities is *successful,* control continues with the statements following the dialog_sequence. The term "success" here means that no defensive, acceptability, or timing checks occurring within the attempt detected an error, and that no exceptions (if the language has exceptions) were propagated out to the attempt's discuss statement. If the attempt was not successful, the process' state is restored from the

```
SELECT
      attempt_1
   OR
      attempt_2
   OR
      attempt_3

   TIMEOUT simple_expression
      sequence _of_statements

   ELSE
      sequence _of_statements
END SELECT;
```

Dialog_Sequence
Figure 4

recovery point and the other attempts will be tried in order. Thus, the dialog_sequence enables the programmer to provide a primary and a list of alternate algorithms by which the process may achieve its goals at that locus of its text.

Exhaustion of all attempts with no success brings control to the **else** part after restoration of the process' state from the recovery point. The **else** part contains a sequence of statements which allows the programming of a "last ditch" algorithm for the process to achieve its goal. If this sequence of statements is successful, control continues after the dialog_sequence. If not, or if there was no statement sequence, the surrounding attempt fails.

Timing constraints can be imposed on colloquys (and hence on dialogs). Any participant in a colloquy can specify a timing constraint which consists of a simple expression on the **timeout** part of the dialog_sequence. Absence of a timing constraint must be made explicit by replacing the simple expression with the keyword **never**. A timing constraint specifies an interval during which the process may execute as many of the attempts as necessary to achieve success in one of them. Should an attempt achieve success

or the list of attempts be exhausted without success before expiration of the interval, further actions are the same as for dialog_sequences without timing specifications. However, if the interval expires, the current attempt fails, the process' state is restored from the recovery point, and execution continues at the sequence of statements in the **timeout** part. The attempts of the other processes in the same dialog also fail but their subsequent actions are determined by their own dialog_sequences. If several participants in a particular colloquy have timing constraints, expiration of one has no effect on the other timing constraints. The various intervals expire in chronological order. As with the else part, the timeout part allows the programming of a "last ditch" algorithm for the process to achieve its goal, and is really a form of forward recovery since its effects will not be undone, at least at this level. If the sequence of statements in the timeout part is successful, control continues after the dialog_sequence. If not, or if there were no statement sequence, the surrounding attempt fails.

In any attempt, a statement sequence (which is logically outside the dialog_sequence) can follow the discuss statement to provide specialized post-processing after the recovery point is discarded if the attempt succeeds. It is not subject to this dialog_sequence's timing constraint.

The programmer is reminded by its position after the **timeout** part that the **else** part is not protected by the timer, and that it is reached only after other (potentially time-consuming) activities have taken place. The structure of the dialog_sequence also requires no acceptance check on these activities. The implication of these two observations is that the last ditch activities need to be programmed very carefully.

A *fail* statement may occur only within a sequence of statements contained within a dialog_sequence. Execution of a fail statement causes the encompassing attempt to fail. The fail statement is intended for checking within an attempt. For example, it can be used to program explicit defensive checks on inputs such as:

```
IF input_variable < lower_bound THEN
   FAIL;
END IF;
```

It can also be used to simplify the logical paths out of an attempt should some internal case analysis reach an "impossible" path. With the fail statement, the programmer does not have to make the code for the attempt complicated by providing jumps or other paths to the acceptance test or to insure that some part of the test is always **false** for such a special path. The fail statement can also be used to provide sequences of statements for the **else** and **timeout** parts that make failure explicit rather than implicit (i.e. failure is indicated by their *absence*).

6. OTHER LANGUAGE FACILITIES

Dialog_sequences can be used to construct deadlines[8], generalized exception handlers[9], recovery blocks, traditional conversations, exchanges[10], and s–conversations[11]. Thus the colloquy is at least as powerful as each of these previously proposed constructs for provision of fault tolerance. For the sake of brevity, we will illustrate only the programming of a recovery block.

A recovery block is a special case of a colloquy in which there is only one process participating, every dialog uses the same acceptance test, there is no timing requirement, and there are no "last ditch" algorithms to prevent propagation of failures of the construct. Figure 5 shows a dialog_sequence that is equivalent to the recovery block shown in Figure 6. The use of the fail statement in the dialog_sequence makes explicit the propagation of the error to a surrounding context just as does the **else error** closing of the recovery block. In the dialog_sequence, the Boolean expression is repeated in the discuss statements rather than gathered into the dialog function because we want to be able to include local variables in it as a programmer of the recovery block would. Should an error be detected in `statement_sequence_1`, the state is restored and `statement_sequence_2` is executed, and so on.

```
FUNCTION abc RETURNS boolean IS BEGIN RETURN TRUE; END abc;
  ....
DIALOG abc SHARES ( );
  ....

SELECT
      DISCUSS abc BY
         statement_sequence_1
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_2
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_3
      TO ARRANGE boolean_expression_1;

   TIMEOUT NEVER;

   ELSE
       FAIL; -- Omitting this line does not change the semantics.
END SELECT;
```

Specification of Colloquy for a Recovery Block
Figure 5

```
ENSURE boolean_expression_1 BY
   statement_sequence_1

ELSE BY
   statement_sequence_2

ELSE BY
   statement_sequence_3

ELSE ERROR;
```

A Recovery Block
Figure 6

Finally, should an error be detected in `statement_sequence_3`, the state is restored and the error is signaled in a surrounding context. An error may be detected by evaluation of

boolean_expression_1 to **false**, or by violation of some underlying interface (such as raising of an exception).

7. CONCLUSIONS

We have introduced a new linguistic construct, the colloquy, which solves the problems identified in the earlier proposal, the conversation. We have shown that the colloquy is at least as powerful as recovery blocks, but it is also as powerful as all the other language facilities proposed for other situations requiring backward error recovery; recovery blocks, deadlines, generalized exception handlers, traditional conversations, s-conversations, and exchanges.

The major features that distinguish the colloquy are:

(1) The inclusion of explicit and general timing constraints. This allows processes to protect themselves against any difficulties in communication that might prevent them from meeting real-time deadlines. It also effectively deals with the problem of deserter processes.

(2) The use of a two-level acceptance test. This allows much more powerful error detection because it allows the tailoring of acceptance tests to specific needs.

(3) The reversal of the order of priority of alternate communication attempts and of recovery points. This allows processes to choose the participants in any alternate algorithms rather than being required to deal with a single set of processes.

(4) A complete and consistent syntax that is presented as extensions to Ada but could be modified and included in any suitable programming language.

Sample programs that have been written (but not executed) using the colloquy show that extensive backward error recovery can be included in these programs simply and elegantly. We are presently implementing these ideas in an experimental Ada testbed.

This paper is not a formal statement of these concepts. The reader may correctly feel that important detail has been omitted. We are only able to present informally the key concepts in a paper of this length. For more details, see [12].

8. ACKNOWLEDGEMENTS

This work was sponsored by NASA grant number NAG1–260 and has been cleared for publication by the sponsoring organization.

REFERENCES

(1) Randell B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, **SE-1**(2), pp. 220–232, June 1975.

(2) *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

(3) Lomet D.B., "Process Structuring, Synchronization and Recovery Using Atomic Actions," *SIGPLAN Notices*, **12**(3), pp. 128–137, March 1977.

(4) Russell D.L., M.J. Tiedeman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, p. 106, June 1979.

(5) Horning J.J., et al., "A Program Structure for Error Detection and Recovery," pp. 171–187 in *Lecture Notes in Computer Science* **Vol. 16**, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin, 1974.

(6) Kim K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering*, **SE-8**(3), pp. 189–197, May 1982.

(7) Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

(8) Campbell R.H., K.H. Horton, G.G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 95–101, 1979.

(9) Salzman E.J., *An Experiment in Producing Highly Reliable Software*, M.Sc. Dissertation, Computing Laboratory, University of Newcastle upon Tyne, 1978.

(10) Anderson T., J.C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering*, **SE-9**(3), pp. 355-364, May 1983.

(11) Jalote P., R.H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," *Digest of Papers FTCS-14: Fourteenth International Conference on Fault-Tolerant Computing*, pp. 347-352, 1984.

(12) Gregory S.T., *Programming Language Facilities for Comprehensive Software Fault-Tolerance in Distributed Real-Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Virginia, 1985.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665 Attention: Mr. Edmond H. Senn ACD, MS 125
4 - 5*	NASA Scientific and Technical Information Facility P. O. Box 8757 Baltimore/Washington International Airport Baltimore, MD 21240
6 - 7	J. C. Knight
8	A. Catlin
9 - 10	E. H. Pancake Clark Hall
11	SEAS files

*One reproducible copy

6435:cms